Block Closures

Marcus Denker, Inria Evref

It looks so simple

```
[ 1 + 2 ] value
```

- Block of code
- It is an object!
- [] creates the object
- #value evaluates it

It looks so simple

```
[ 1 + 2 ] value
```

Is it not just a wrapper around a method implementing #value?

But it is not

```
| temp myBlock |

temp := 1.
myBlock := [ 1 + temp].
^  myBlock
```

- The block can access temps from the outer method (or block)
- The block could access self and ivars
- the block can survive the execution of the method where the temp is defined

A block is code

- Thus: is not just a method?
- Indeed: CompiledBlock
  - Shares the Superclass with CompileMethod
  - Very similar: CompiledMethodLayout
    - Bytecode
    - Literals

Lets look at the Bytecode

- Lets look at a simple method with a simple block:

```
myMethod
^ [1 + 2]
```

Bock creation

- CompiledBlock instance is part of the literals
- A bytecode creates an instance of FullBlockClosure
    - this happens at runtime (!)
    - When the [] is evaluated
- But why?

Creation vs Activation

- This is a bit hard to wrap your head around, thus we repeat:

```
[ … ]
```

Block Creation: VM creates instance of FullBlockClosure

```
aBlock value
```

Block Activation: the VM executes the CompledBlock

Excursion: Local Vars

- Imagine a method with local variables

```
| temp1 temp2 |

temp1 := 1.
temp2 := 2.
^temp1 + temp2
```

Where are temp1 and temp2 stored?

Executing a Method

- I send a message to the object
- VM looks up the selector in the class hierarchy
- lookup finds a CompiledMethod
- Creates a StackFrame (think of it like an Array)
    - every temp has an offset here

No manual cleanup

- After method finishes execution, GC cleans up
- If no other reference to an object referenced from a temp ==> object gets garbage collected
- No need for you to manually manage it !

Back to the closure

- And our example with an “escaping” var

- A block can reference a temp from the outer method!

```
| temp myBlock |

temp := 1.
myBlock := [ 1 + temp].
^  myBlock
```

“Escaping” Vars

- We need a way to access the Variables
  - read
  - write
- And there can be multiple blocks!
  - we need to have access to the one place where the var is stored

Temps are in Contexts

- Conceptually, the temp is stored in the Context where it is defined
- In the example, we read temp from the method context

```
| temp myBlock |

temp := 1.
myBlock := [ 1 + temp].
^  myBlock
```

But the block lives longer than the method context!

Naive idea: keep contexts

- We could just access the temps via the definition contexts
- Lots of problems
    - Slow (but special bytecode could help)
        - Especially as Contexts are created on demand
        - All contexts would stay alive till the *block* is GCed
            - all objects referenced by the temps
            - receiver (self) of the method where the block is defined

What else can we do?

- Two cases: Read and Write

- Rethink the place where temps are stored

Two cases: read and write

- The closure could just read the escaping var

```
| temp myBlock |

myBlock := [ 1 + temp].
^  myBlock
```

- The closure could write the escaping var

```
| temp myBlock |

myBlock := [:arg | temp := arg].
^  myBlock
```

Escaping Read

- The variable is not changed in a closure, just read
- We just need to get a reference to it to read it
- Solution: use the Stack!
  - “Closure Conversion” -> turn escaping var access into a (hidden) additional parameter.
- Push in stack before closure creation
  - Closure has a hidden local temp with it

Escaping Write

- So we really need to use the definition context?
- We could just have another object (an Array)
- both defining method *and* closure get a reference to the array
- They use at:put: to write

The TempVector

- This is what the TempVector is all about
- one Array per method / block for all defined temps that escape *and* have one write.
- We push it on the stack (just like a copied var)
- Special read/write bytecodes for speed

The Bytecode

- Lets look at some examples

Why do we create block at runtime?

- We need access to the temporary variables from outer blocks / the method
- We push them on the stack (copied vars)
- We create tempVectors and pas those as copied vars via the stack
- And: We need access to self (for ivar access, too)

Some Optimizations

- Creating blocks is slow
- What can we optimize?

Clean Blocks

- What if a block does not access any variables from an outer method ?
- And no self
  - and no ivars (as we need self to read the ivar)
- And no return (as it needs the home context)

Pre-create Clean Blocks

- We do not need any information from runtime
- We can create the block at compile time
- And store the block in the literal frame, not the CompiledBlock
- A clean block is exactly the “naive” block idea we had at the start: it’s just a wrapper around a method that implements #value methods

What about constants?

- An even more trivial block: a block that just returns a literal
- Happens more often than you think!
- e.g. at:IfAbsent:

```
Morph>>#minHeight
    "answer the receiver's minHeight"
    ^ self
        valueOfProperty: #minHeight
        ifAbsent: [2]
```

IWST Talk!

- CleanBlocks: Faster [] block creation
- ConstantBlocks: CleanBlocks where we can speed up #value
- Come to out IWST presentation to learn more!

Problem TempVector

```
| temp1 temp2 myBlock |
temp1 = HugeObject new.
temp2 = #hello.

[temp1 := 1] value.
^  [ temp2 := 1]
```

- We have one tempVector per method
- Leads to memory leaks!
- The long living block references just temp2
    - but via tempVector which holds onto the HugeObject

Idea TempVector Splitting

- We could partition TempVector according to liveness
  - in some way similar to register allocation
- **But**: trade-off of “lots of temp vectors” vs “overlapping liveness”
- First step: analyse how often this happens

Enable Clean for real

- Enable Clean Blocks by default
- Harder than you would think
  - Lots of code assumed that there is an outerContext
  - #home can be found via outerContext
- e.g. rewrite homeMethod to use static outerCode, not “homeContext method”

Problem: outerContext

- block is created with reference to outerContext
    - which is either the home context (method) or another block Context

- Slow (even with optimization of context object just created on access)
- Lots of data can not be GCed
- Defeats to some extend the tempVector/copiedVars optimization

Block with no outer Context

- The block creation bytecode supports creating blocks without outer context
- We need it just for return
- End even here: we need the homeContext, not the outer context
    - Experiment with closure model that references home directly (if needed)

If you want to help

- Enable clean blocks by default
- Enable block without outer context
- TempVector splitting
- Experiment: Fullblocks referencing home instead of outer